1. CPU结构和功能

14.1 CPU组成

14.2 寄存器组成

用户可见寄存器

控制和状态寄存器

微处理器寄存器组成的例子

14.3 指令周期

间接周期

数据流

14.4 指令流水线技术

流水线策略

流水线性能

流水线冒险

处理分支指令

Intel80486的流水线

14.5 x86系列处理器

寄存器组成

中断处理

14.6 ARM处理器

处理器组成

处理器模式

寄存器组成

中断处理

14.7 关键词、思考题和习题

**学习目标**

学习本章后，你应该能够：

* 区分**用户可见寄存器**和**控制/状态寄存器**，并讨论每个类别中寄存器的用途。
* 总结**指令周期**
* 讨论**指令流水线**的原理以及它在实践中是如何工作的。
* 比较和对比各种形式的管道危险
* 概述x86处理器结构。
* 概述ARM处理器结构

本章继续第三部分中未完成的讨论，并为在第15章和第16章中讨论RISC和超标量结构奠定了基础。

  本章以CPU组成为开始，然后分析构成处理器内部存储器的寄存器，接着再返回到指令周期的讨论（开始于3.2节）。在那里将完整地说明指令周期和被称为指令流水线的通用技术。最后以考察x86和ARM处理器组成的各方面情况来结束本章。

**14.1CPU组成**

为了理解CPU的组成，让我们考虑对CPU的要求，它必须完成以下任务：

■**取指令**：CPU从存储器（寄存器、cache、主存）读取指令。

■**解释指令**：必须对指令进行译码，以确定所要求的动作。

■**取数据**：执行指令可能要求从存储器或输入/输出（I/O）模块读取数据。

■**处理数据**：执行指令可能要求对数据完成一些算术或逻辑操作。

■**写数据**：执行结果可能要求将数据写到存储器或I/O模块。

显然，为了能做这些事情，CPU需要暂时存储某些数据。CPU必须记住当前执行的指令的位置，以便知道从何处得到下一条指令。CPU还需要在执行指令期间，暂时保存指令和数据。换句话说，CPU需要一个小的内部存储器。

图14.1是一个CPU的简化视图，指出了它通过系统总线与系统其余部分的连接。对第3章所描述的任一种互连结构，都需要一个类似的接口。 读者会回忆起，CPU的主要部件是一个算术逻辑单元（ALU）和一个控制器（CU）。ALU进行数据的实际计算或处理。控制器控制数据和指令移入移出CPU，并控制ALU的操作。此外，该图还表示了由一组存储位置组成的极小的内部存储器，称为寄存器。

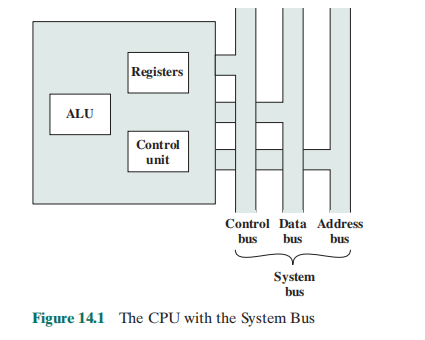
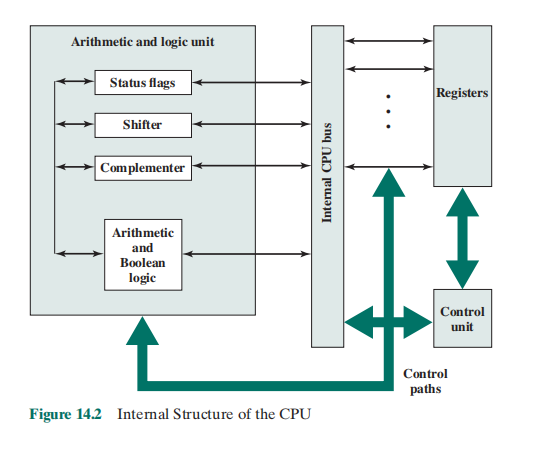


图14.2是一个更详细的CPU视图，指示数据传送和逻辑控制的路径，包括一个标记为“CPU内部总线”的组件。需要有这么一个组件在各种寄存器和ALU之间传输数据，因为ALU实际上只对CPU内部存储器中的数据进行操作。此图还表示了ALU典型的基本组件。注意计算机整体的内部结构与CPU的内部结构之间的相似性。两种结构中都有一个主要组件的小集合（计算机的CPU、I/O、存储器，CPU的控制器、ALU、寄存器）通过数据通路连接在一起。



**14.2寄存器组成**

正如第4章所述，计算机系统采用了存储器分级系统。此分级系统的级别越高，存储器越快、越小，也越昂贵（每位）。CPU内有一组寄存器，它们的存储器级别在分级系统中位于主存和cache之上。CPU中的寄存器可分为两类：

* **用户可见寄存器（user-visible register）**：允许机器语言或汇编语言的编程人员通过优化寄存器的使用而减少对主存的访问。
* **控制和状态寄存器（control and status register**）：由控制器来控制CPU的操作，并由拥有特权的操作系统程序来控制程序的执行。

这两类寄存器的划分界限并不明确。例如，在某些机器上程序计数器是一个用户可见寄存器（如x86），但在很多机器上却不是。为了顺利进行下面的讨论，我们将采用上述分类。

**用户可见寄存器**

 用户可见寄存器是指可通过机器语言方式访问的寄存器。这些用户可见寄存器可分为：通用、数据、地址、条件码。

**通用寄存器（general-purpose register）**可被程序员指派各种用途。有时，它们在指令集中的使用是正交于操作的，即任何通用寄存器能为任何操作码容纳操作数。这展现了真正通用的意义。然而，常常不是这样的，而是有某些限制。例如，可能有专用于浮点操作和栈操作的通用寄存器。

某些情况下，通用寄存器可用作寻址功能〔如寄存器间接寻址、偏移寻址）。在其他情况下，数据寄存器（data register）和地址寄存器（address register）之间有部分或完全的区分。**数据寄存器**仅可用于保持数据而不能用于操作数地址的计算。**地址寄存器**可以是自身有某些通用性，或是专用于某种具体的寻址方式。例如：

* **段指针（segment pointer）**：在提供分段寻址的机器中（见8.3节），段寄存器保持着该段的基地址。可以有多个段寄存器，例如，一个是操作系统的，一个是当前进程的。
* **变址寄存器 （index register）**： 这些寄存器用于变址寻址，并可能是自动变址的。
* **栈指针（stack pointer）**： 若有用户可见的栈寻址方式，则一般来说栈会被分配在存储器中，而CPU内有一专用的寄存器指向栈顶。这允许隐含寻址，即push、pop和其他不需要显式地指定钱操作数的栈指令。

这里有几个设计出发点。重要的一点是，使用完全通用的寄存器还是规定各寄存器的用途。在上一章我们已接触到这个问题，因为它影响指令集的设计。对专用寄存器的使用，一个操作数指定符所引用的寄存器类型通常能隐含在操作码中。操作数指定符必须做的只是标识这一组专用寄存器中的某一个将被使用，而不是所有寄存器中的某一个．于是节省了位数。另一方面．这种规定又限制了程序员的灵活性。

另一设计出发点是寄存器数量。同样，这影响指令集的设计，因为寄存器越多，需要的操作数指定符的位数也越多。正如前面所讨论的，某些机器有8到32个寄存器是适宜的［LUND77］。太少的寄存器会导致更多的存储器访问，太多的寄存器又不能显著地减少存储器访间（如 [WILL90]）。然而，一种新的方法使得在某些RISC系统中展示出了使用上百个寄存器的优点，这些将在第15章讨论。

最后，这里还有一个寄存器长度问题。用于保存地址的寄存器明显要求其长度足以容纳最长的地址。数据寄存器应能保存大多数数据类型的值。某些机器允许两个相邻的寄存器作为一个寄存器来保持两倍长度的值。

最后一类寄存器是用于保存**条件码（condition code）**， 亦称为标志（flag）的寄存器，它们至少是部分用户可见的。CPU硬件设置这些条件位作为操作的结果。例如，一个算术运算可能产生一个正的、负的、零或溢出的结果。除结果本身存于寄存器或存储器之外，一个条件码亦相应被设置。这些条件码可被后面的条件分支指令所测试。

条件码通常被收集到一个或多个寄存器中。通常，它们构成控制寄存器的一部分。机器指令允许这些位以隐含引用的方式读出，但它们不能被程序员更改。

许多处理器，包括那些基于 IA-64 体系结构的处理器和 MIPS 处理器。根本不使用条件码。相反．它们采用测试条件分支指令．这种指令指定一种比较操作，并根据比较的结果产生控制转移动作不保存条件码。基于[DER087]的表14-1列出了条件码的主要优缺点。

表14-1 条件码

|  |  |
| --- | --- |
| **优点** | **缺点** |
| 1.因条件码由常规的算术和数据传送指令建立，故它们能减少对比较和测试类指令的需求。  2.条件指令，例如“BRANCH”这样的条件分支指令要比测试条件分支（TEST AND BRANCH）这样的复合指令简单。  3.条件码便利了多路分支选择。例如，一条测试指令之后可带两条分支，一条按小于或等于0的条件码进行，另一条按大于0进行。  4.条件代码可以在子程序调用期间与其他寄存器信息一起保存在堆栈中。 | 1.条件码增加了硬软件的复杂性。条件码位被不同的指令以不同的方式频繁修改，使微程序编程人员和编译器设计人员的工作更为因难。  2.条件码不正规，通常它们不是主数据路径的一部分，故要求额外的硬件连接。  3.使用条件码的机器经常需要为诸如位检查、循环控制、原子式信号量操作等这类特殊情况添加专门的非条件码指令。  4.在流水实现中，条件码要求专门的同步化，以避免冲突。 |

某些机器中，一个子程序调用将导致自动保存所有用户可见的寄存器，在返回时自动取回。这些保存和恢复是作为调用和返回指令执行功能的一部分由CPU完成的。这就允许各个子程序独立地使用用户可见寄存器。而在其他一些机器上，子程序调用之前保存相关用户可见寄存器的内容是程序员的责任，他们要在程序中为此专门安排一些指令。

**控制和状态寄存器**

有一类寄存器在CPU中起着控制操作的作用。它们中的大多数，在大多数机器上，是用户不可见的。某些在控制或操作系统模式下执行的机器指令是用户可见的。

当然，不同的机器将有不同的寄存器组织并使用不同的术语。这里，我们列出一个相对完整的寄存器类型列表，并予以简短描述。

对于指令执行，有4种寄存器是至关重要的。

* + **程序计数器（PC）**：存有待取指令的地址。
  + **指令寄存器（IR）**：存有最近取来的指令。
  + **存储器地址寄存器（MAR）**：存有存储器位置的地址。
  + **存储器缓冲寄存器（MBR）**：存有将被写入存储器的数据字或最近从存储器读出的字。

不是所有的处理器都有专门称为MAR和MBR的寄存器，不过还是会有某种等价的缓冲机制，其中要被写到系统总线上的数据位，以及从系统总线上读到的数据位，都会被暂时保留或存储起来。

通常，在每次取指令之后， PC的内容即被CPU更改，故它总是指向将被执行的下一条指令。转移或跳步指令亦修改PC的内容。取来的指令装入IR，在那里分析操作码和操作数指定符。与存储器的数据交换使用MAR和MBR。在总线组织的系统中，MAR直接与地址总线相连，MBR直接与数据总线相连。然后，用户可见寄存器再与MBR交换数据。

刚才提到的4个寄存器用于CPU和存储器之间的数据传送。在CPU内，数据必须提交给ALU来处理。ALU可对MBR和用户可见寄存器直接存取。相应地也可在ALU的边界上有另外的缓冲寄存器，这些寄存器能作为ALU的输入和输出，可与MBR和用户可见的寄存器交换数据。

很多CPU设计都包括常称为程序状态字（program status word， PSW）的一个或一组寄存器。PSW一般含有条件码加上其他状态信息。通常PSW包括下列字段或标志：

* **符号（sign）**：容纳最后算术运算结果的符号位。
* **零（zero）**：当结果是0时被置位。
* **进位（carry）**：若操作导致最高位有向上的进位（加法）或借位（减法）时被置位。用于多字算术运算。
* **等于（equal）**：若逻辑比较的结果相等，则置位。
* **溢出（overflow）**：用于指示算术溢出。
* **中断允许/禁止**：用于允许或禁止中断。
* **监管（supervisor）**：指出CPU是执行在监管模式中还是在用户模式中。某些特权的指令只能在监管模式中执行，某些存储器区域也只能在监管模式中被访问。

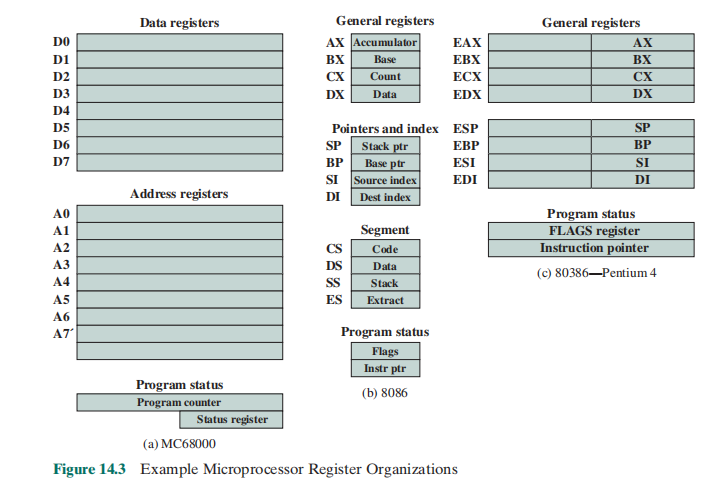
一些具体CPU设计中可能还会有其他额外的有关状态和控制的寄存器。例如，除了PSW之外，可能有一个指向存储器块（例如进程控制块PCB）的指针寄存器，而此存储块含有附加的状态信息。在使用向量式中断的机器中，可能提供有一个中断向量寄存器。若栈用于实现某些功能（例如子程序调用），则需要有一个系统栈指针。对于虚拟存储器系统，可能会有一个页表指针寄存器。最后，在I/O操作控制方面也可能需要有专门的寄存器。

设计控制和状态寄存器组织时有几个因素需考虑。一个关键的考虑是对操作系统的支持。某些类型的控制信息是专门为操作系统使用的。若CPU设计者对将要使用的操作系统有基本的了解，则寄存器的组织可能在一定程度上为该操作系统定制。

另一个关键的考虑是控制信息在寄存器和存储器之间的分配。一种普遍的做法是将存储器最前面（最低地址）的几百或几千个字用于控制目的。设计者必须决定多少控制信息应在寄存器中，多少应在存储器中。这通常要在成本和速度之间进行权衡。

**微处理器寄存器组成的例子**

考察和比较一些可比系统的寄存器组成是有指导意义的。本节我们考察大约在同一时期设计出的2个16位微处理器： Motorola MC68000 [STRI79]和Intel 8086 [MORS78]。图14-3a和图14-3b分别给出了上述两种处理器的寄存器组成，像存储器地址寄存器这样的纯内部寄存器未在图中示出。



MC68000将它的32位寄存器分成8个数据寄存器和9个地址寄存器。8个数据寄存器主要用于数据操作，并在寻址方式中用作变址寄存器。寄存器的宽度允许8位、16位或32位数据操作，具体取决于操作码。地址寄存器包含32位（不分段）地址，其中两个寄存器亦用作栈指针，一个用于操作系统，一个用于用户，这取决于当前的执行模式。这两个寄存器都编号为7，因为任何时刻只能一个在使用。MC68000还包括一个32位程序计数器和一个16位状态寄存器。

Motorola设计小组也希望有一个很规整的指令集而不带有专门的目的寄存器。对代码效率的考虑使他们将寄存器分成两个功能组件，在每个寄存器指定符上节省了1位。这看起来是在完全通用性和代码紧凑性之间的一个合理折中。

Intel 8086采取另一种不同的方法来组织寄存器，每个寄存器都有专门的用途，虽然某些寄存器也可通用。8086含有4个16位数据寄存器，它们亦可按字节（8位）来使用；还含有4个16位的指针和变址寄存器。数据寄存器在某些指令中可用作通用寄存器，而在另一些指令中它们是隐含被使用的。例如，乘法指令总是使用累加器。4个指针寄存器亦是在几种操作中隐含被使用，每个用于保存段内位移。8086还有4个段寄存器，其中三个以一种专门的隐含方式来使用，分别指向一个包含当前指令的段（对转移指令特别有用），一个包含数据的段，一个包含栈的段。这些专门的隐含方式的使用，以减少灵活性为代价，提供了编码的紧凑性。8086还包括一个指令指针寄存器和一组状态和控制标志，其中每个状态和控制标志都是1位。

通过这个比较可以看清楚的一点是，到目前为止，关于组织CPU寄存器的最好方式还没有一个普遍接受的原则[TOON81]。正如对整个指令集设计的情况一样，也有众多的CPU设计观点，这些都还是有待品评的事情。

关于寄存器组织设计的第二个有指导意义的观点说明如图14-3c所示。此图表示了Intel 80386 [ELAY85]的用户可见寄存器的组织。80386是32位微处理器，并设计成8086的扩展。80386使用32位寄存器。然而，为了向在早先机器上写成的程序提供向上兼容，80386将原先的寄存器组织嵌入到新组织中。给定这种设计限制，这个32位微处理器的寄存器组织设计明显在灵活性上受到制约。

**14.3指令周期**

前面3.2节已描述过CPU的指令周期（参见图3-9）。指令周期包括如下子周期：

* + **取指（fetch）**：将下一条指令由存储器读入CPU。
  + **执行（execute）**：解释操作码并完成指定的操作。
  + **中断（interrupt）**：若中断是允许的并且有中断发生，则保存当前进程的状态并为此中断服务。

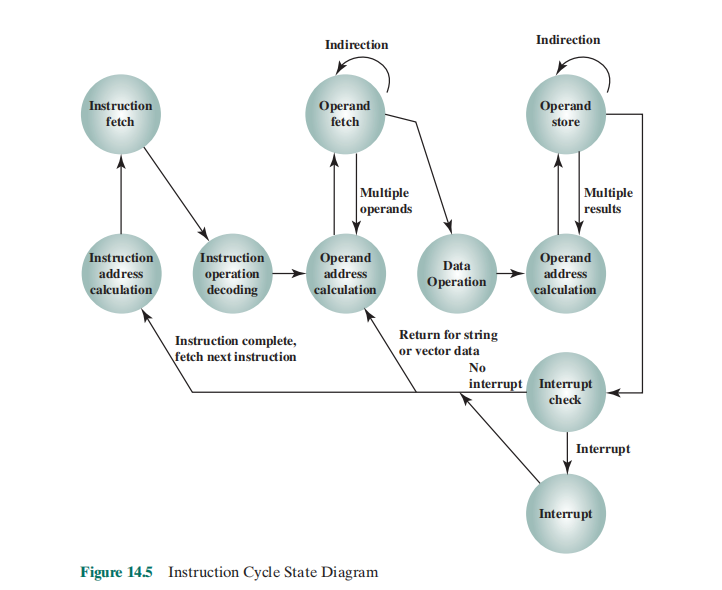
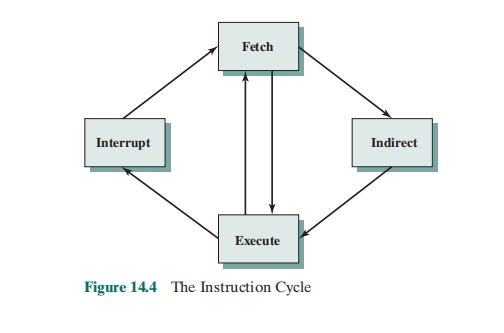
现在我们来详细描述指令周期。首先，必须引入一个另外的子周期，称为间接周期（indi-

rect cycle）。

**间接周期**

在第13章我们已看到，指令的执行可能涉及一个或多个存储器中的操作数，它们每个都要求一次存储器访问。而且，若使用间接寻址，则还需要额外的存储器访问。

可把间接地址的读取看成是一个额外的指令子周期，其过程显示于图14-4，动作的主线由交替的取指令和指令执行动作组成。取来一条指令之后，要对它进行检查以确定是否需要间接寻址。如果是，则所要求的操作数使用间接寻址方式取来。在执行之后，可能有一个中断在取下一条指令之前被处理。

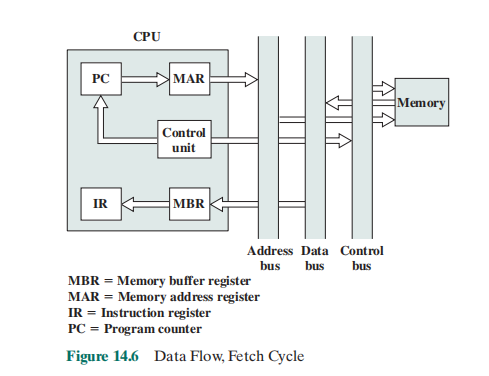


观察此过程的另一种方式如图14-5所示，它是图3-12的改进版。它更准确地说明了指令周期的实质。一旦取来一条指令，它的操作数指定符必须被识别。然后读取存储器中的每个操作数，这个过程可能要求间接寻址。寄存器操作数不需要从存储器读取。一旦操作被执行，可能需要一个类似的过程将结果存人主存。

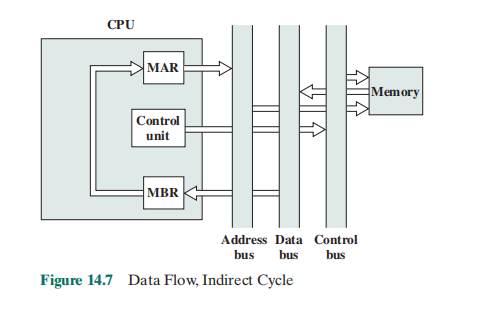
**数据流**

指令周期期间，所发生事件的实际序列取决于CPU的具体设计。不过我们还是可以从一般意义上列出哪些事件是应该要发生的。假定一个CPU有一个存储器地址寄存器（MAR）、一个存储器缓冲寄存器（MBR）、一个程序计数器（PC）和一个指令寄存器（IR）。

在取指周期，一条指令由存储器读入，图14-6显示了此期间的数据流动。开始时， PC存有待取的下一条指令的地址。这个地址被传送到MAR并放到地址总线上。控制器发出一个存储器读的请求，存储器把结果放到数据总线上，CPU将其复制到MBR，然后传送到IR。在此期间，PC增1，为下次取指令做好准备。

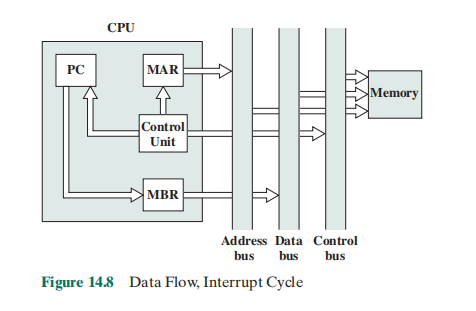


一旦经历取指周期，控制器检查IR的内容，以确定是否有一个使用间接寻址的操作数指定符。若是，则进入间接周期，如图14-7所示，这是一个简单周期。MBR最右边的N位是一个地址引用，被传送到MAR。然后，控制器请求一个存储器读，得到所要求的操作数地址，并送人MBR。



取指和间接周期是简单的，且是可预期的。执行周期则有多种表现形式，具体是哪种形式取决于各种不同的指令里哪一条当前在IR中。这个周期可能涉及寄存器间的数据传送，对存储器或I/O设备的读或写，以及ALU的功能使用。

像取指和间接周期一样，中断周期是简单的并可预期的（见图14-8）。PC的当前内容必须被保存，以便在中断之后CPU能恢复先前的动作。于是，PC的内容传送到MBR，将被写入存储器。为此目的，一个专门的存储器位置被控制器装入MAR，它可能是一个栈指针。随后，中断子程序的地址装入PC。结果是，下一指令周期将以取此相应的指令而开始。



**14.4 指令流水线技术**

随着计算机系统的发展，特别是集成电路工艺上的改进，例如更快的电路，可以实现更高的性能。另外， CPU组织的改进也能改善性能。我们已经看到过这样的例子，如使用多个寄存器而不是单一的累加器，又如使用高速缓存等。另外一种使用非常普遍的组织方法是指令流水线技术。

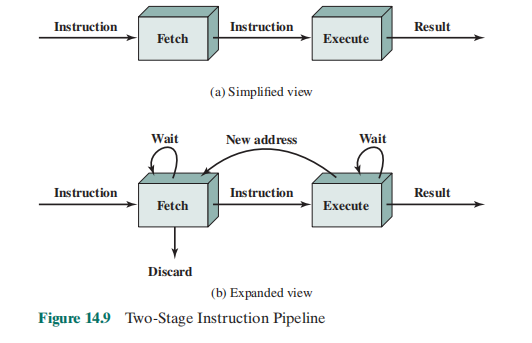
**流水线策略**

指令流水线类似于工厂中装配线的使用。装配线利用了这样一个事实，即一个产品要经过几个制作步骤。通过把制作过程安排在一条装配线上，多个产品能在各个阶段同时被加工。

这种过程称为流水处理（pipelining），因为在一条流水线上，当先前接收的输入已成为加工的结果出现在另一端时，新的输入又在一端被接收进来。

将这种概念施加到指令的执行上，我们必须认识到，事实上一条指令的执行也是分成几个步骤。图14-5就是一个例子，它将指令周期分成10个顺序的任务。很清楚，这里应有实施流水线技术的某种机会。

作为一种简化的方法，考虑将指令处理分成两个阶段：取指令和执行指令。在一条指令执行期间，主存可能没有存取的操作。此时主存能用于取下一条指令，从而这个取指操作与当前指令的执行并行工作。图14-9a描述了这种方法。此流水线有两个独立的阶段。第一个阶段取一条指令并缓存它，当第二个阶段空闲时，将第一个阶段缓存的指令输送给它。当第二个阶段正在执行此指令时，第一个阶段利用未使用的存储器周期读取下一条指令并缓存它。这称为指令预取（instruction prefetch）或取指交叠（fetch overlap）。注意这种方式涉及指令缓存（instruction buffering），因此需要更多的寄存器。一般来说，流水处理都需要额外的寄存器用以在流水段之间保存数据。



显然这种处理将加快指令的执行。若取指和执行这两个阶段是相等的时间，则指令周期时间将是原来的一半。然而，若我们更仔细地查看这个流水线（见图14-9b），将会看到实现这种执行速度的翻倍是不太可能的，理由有二：

（1）执行时间一般要长于取指时间。执行将涉及读取和保存操作数以及完成某些操作。于是，取指阶段可能必须等待一定的时间才能排空它的缓冲器。

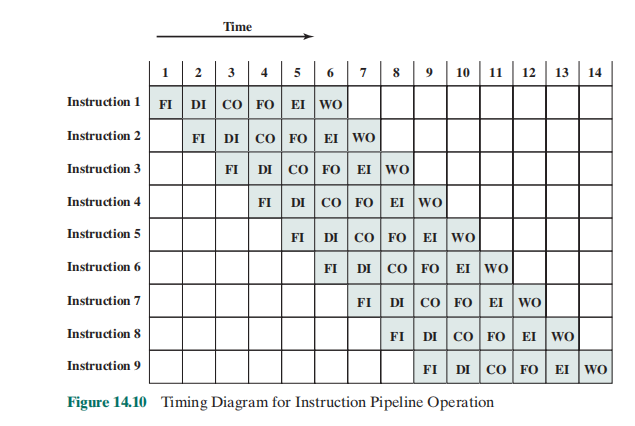
（2）条件分支指令使得待取的下一条指令的地址是未知的。于是，取指阶段必须等待，直到它能由执行阶段得到下一条指令地址。而在取下一条指令时执行阶段又可能必须等待。

由第二种情况造成的时间损失可通过推测来减少。一个简单的规则如下：当一条条件分支指令由取指阶段传送到执行阶段时，取指阶段读取存储器中此分支指令之后的指令。于是，若转移未发生，则没有时间损失；若转移发生，则已读取的指令要作废，并读取新的指令。

虽然这些因素降低了两段流水线的潜在效率，但还是带来某种加速。为获得进一步的加速流水线必须有更多的阶段。让我们考虑指令处理的如下分解。

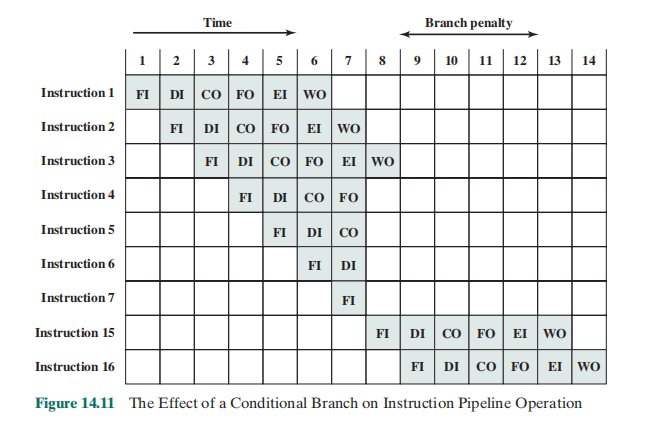
* **取指令（FI）**：读下一条预期的指令到缓冲器。
* **译码指令（DI）** ：确定操作码和操作数指定符。
* **计算操作数（CO）**：计算每个源操作数的有效地址，这可能涉及偏移寻址、寄存器间按寻址、间接寻址或其他形式的地址计算。
* **取操作数（FO）**：从存储器取出每个操作数。寄存器中的操作数不需要取。
* **执行指令（EI）**：完成指定的操作。若有指定的目的操作数位置，则将结果写入此位置。
* **写操作数（WO）**：将结果存入存储器。

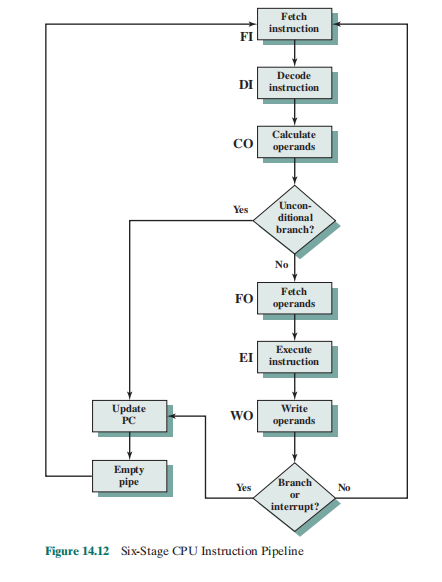
按照这种分解，各个阶段所需要的时间几乎是相等的。为便于说明，假定是相等的时间。图14-10则表示了一个这样的6段流水线，它能将9条指令的执行时间由54个时间单位减少到14个时间单位。



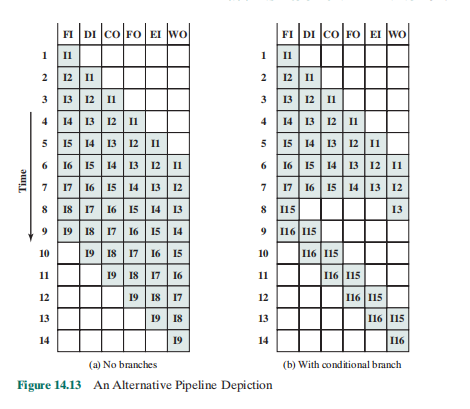
以下是几点说明：此图假设每条指令都通过流水线的6个阶段，但并不总是这种情况。例如，一条LOAD指令就不需要WO阶段。然而，为简化流水线硬件设计，就在假定每条指令都要求这6个阶段的基础上来建立时序。还有，此图是假定所有阶段都能并行完成，具体地说，是假定没有存储器冲突。如FI， FO和WO都涉及存储器访问，此图暗示它们是能同时进行的。大多数存储器系统不准许这样，然而可能所要求的值在cache中，或者FO或WO阶段是个空操作。于是，多数情况下，这种存储器冲突并不会减慢流水处理速度。

有几个因素限制了性能提升。若6个阶段不全是相等的时间，则正如我们在前面讨论的两阶段流水线那样，会在各个流水阶段涉及某种等待。另一困难是条件转移指令，它能使若干指令的读取变为无效。另一不可预料的事件是中断。图14-11说明了条件转移的影响，其中使用与图14-10同样的程序。假定，指令3是一个可能转到指令15的条件转移指令。直到指令执行之前，没办法知道下一条指令到底是哪条指令。此例中，流水线只是简单地按顺序装入下一条指令（指令4）并继续执行。图14-10表示转移未发生，我们得到了全面的性能提升。图14-11表示的是转移发生的情况，但这直到时间单位7结束时才能确定。此时流水线必须清除那些已取来的无用指令。这样，在时间单位8，指令15进入流水线。在时间单位9-12期间没有指令完成，这是由于我们不能预测转移是否发生所导致的性能下降。图14-12指出考虑到转移和中断，流水线所需要的逻辑。





还有一些问题，它们不出现在简单的两阶段流水线中。CO阶段可能需要某个寄存器的内容，而此值可能被仍在流水线中的先前指令所修改。其他的寄存器和存储器冲突也可能出现。系统必须考虑到处理这类冲突的逻辑。



为清晰起见，换一种方式来查看流水线操作将是有益的。图14-10和图14-11是水平方向表示时间，每行表示一条指令的执行。新给出的图14-13是垂直向下表示时间的进行，而每一行表示的是给定时间点的流水线状态。在图14-13a （对应于图14-10）中，时间6时流水线已满，有6条不同的指令正在流水线各阶段进行，并一直保持满负荷运行，直到时间9；假定I9是最后一条待执行指令。在图14-13b （对应于图14-11）中，时间6和7时流水线是满的。在时间7，一条转移指令I3正处于执行阶段，它将引发转移到指令I15。此时，指令I4~I7都要由流水线中被清除出去，于是，时间8时只有I3和I15两条指令在流水线中。

从前面的讨论，我们可能会认为流水线中阶段数目越多，执行速度越快。但IBM S/360的设计者指出，有两个因素将使这种看似简单的高性能设计失败[ANDE67a]，而且这些观点至今仍是有效的。

（1）流水线的每一阶段，都会有某些开销涉及数据在缓冲器间的传送，以及涉及完成各种准备和递交功能。这些开销能明显地使单一指令总的执行时间变长。当顺序指令之间发生逻辑依赖（这种逻辑依赖或是因为大量使用了分支指令，或是因为存储器访问的相关性）时，这个问题就变得特别严重。

（2）优化流水线的使用和处理存储器及寄存器相关性所需的控制逻辑总量，会随着流水线阶段数的增长而急剧增长。这将导致这样一种情形，阶段之间的门控逻辑比这些阶段本身的逻辑还要复杂。

另一个要考虑的因素是锁存延迟（latching delay），即流水线阶段之间的缓冲需要一定时间来完成其操作，而这会增加指令周期的时间。

指令流水线是一种提高性能的强有力技术，但需要精心设计，以合理的复杂性达到最优的效果。

**流水线性能**

下面我们对流水线性能和相对加速予以简单度量（基于[HWAN93]中的讨论）。**指令流水线**的周期时间，是在流水线中将一组指令推进一段所需的时间，图14-10和图14-11中的每一列都代表一个周期时间，周期时间能表示成：



其中：

= 流水线第段的电路延迟时间

= 最大段延迟（通过耗时最长段的延迟）

= 指令流水线段数

= 锁存延时（数据和信号从上一段送到下一段所需的段间锁存接收时间）

通常，延时d等于时钟脉冲的宽度并且。现假设有n条指令在进行，无转移发生。令为阶段流水线执行所有n条指令所需的总时间，则有：



完成第1条指令执行需要个周期。完成其余n-1条指令需要n-1个周期。这个等式很容易由图14-10得到验证，第9条指令在周期14时完成，于是有：



现在考虑一个具有等效函数但没有流水线的处理器，并假设指令周期时间为。使用指令流水线相对于不使用流水线的加速比定义为：

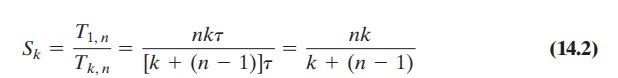
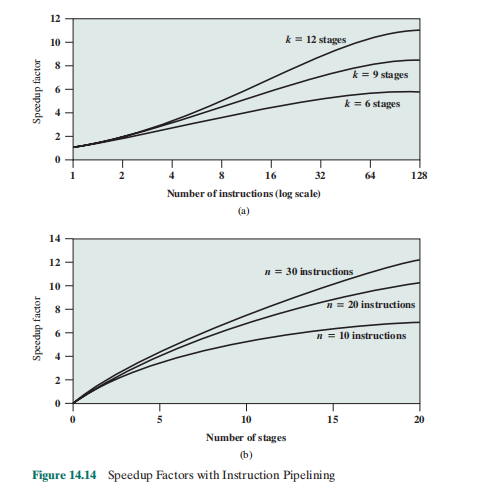


图14-14a给出无分支情况下加速比随指令数的变化关系，当时，，即我们获得k倍加速。图14-14b给出加速比随指令流水线段数的变化关系。在这种情况下，加速比能接近进入流水线而且无转移的指令数。于是，更多的流水线段数能带来更大的潜在加速比。然而，增加更多的段所带来的增益，必须考虑到成本的增加、段间延时的增多，以及遇到转移指令而要求清空流水线的这些事实。



**流水线冒险**

在前一小节，我们提到某些情况会导致流水线不能达到理想加速比。在本小节中，我们将更加系统地考察这个问题。第16章介绍了超标量流水线组织中的复杂性后，也将更细致地回顾这个问题。

**流水线冒险（pipeline hazard）**发生在流水线或流水线的某个部分，因为某些条件不允许流水线继续运行，而必须停顿（stall）的时候。这样的流水线停顿也通常被称为流水线空泡（pipeline bubble）。有3种类型的流水线冒险：资源冒险、数据冒险和控制冒险。

**1.资源冒险**

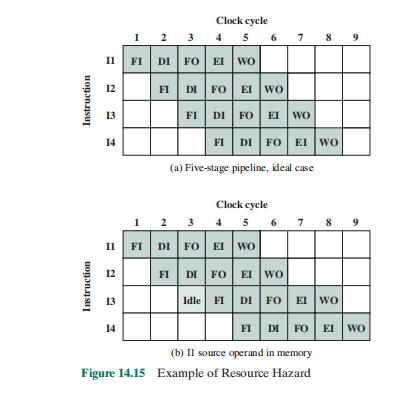
资源冒险（resource hazard）发生在两条（或多条）已进入流水线的指令需要使用相同资源的时候。结果就是在流水线某个部分，这些指令必须串行执行，而不是并行执行。资源冒险在有些时候也称为结构冒险（structural hazard）。

让我们考察一个简单的资源冒险例子。假设有一个简单的5阶段流水线，其中每个阶段的执行时间是一个时钟周期。图14-15a显示了理想的运行情况，其中每个时钟周期都有一条新指令进入流水线。现在假设主存只有一个端口，所有的指令读取以及数据装载和保存都一次只能有一个单独操作。另外，假设没有高速缓存。在这种情况下，从内存装载一个操作数，或向内存保存一个操作数，与从内存读取指令是不能同时进行的。图14-15b显示了这种情况的例子，其中假设指令I1的源操作数在内存中而不是在寄存器中。这样，流水线的取指阶段在进行指令I3的读取之前，必须空闲一个时钟周期。图中假设其他操作数都在寄存器中。

另一个资源冲突的例子是当多条指令可以进入执行阶段，但只有一个算术逻辑单元的时候发生的情况。资源冒险的一种解决办法是增加可用的资源，例如提供访问主存的多个端口，以及提供多个算术逻辑单元。



一种分析资源冲突，辅助流水线设计的方法是预约表（reservation table）。我们会在附录N中考察预约表。



**2数据冒险**

数据冒险（data hazard）发生在对一个操作数位置的访问出现冲突的时候。通常可以用如下形式来描述数据冒险：程序中的两条指令依次执行，并且都将访问同一个内存或寄存器操作数。如果两条指令的执行是严格串行的，那么没有问题发生。但是，如果这两条指令在流水线中运行，那么有可能操作数会不按次序更新，从而导致与严格串行执行不一样的结果。换句话说，就是由于使用了流水线，导致程序运行产生了不正确的结果。

作为一个例子，考虑下面的x86机器指令序列：

ADD EAX,EBX /\*EAX=EAX+EBX

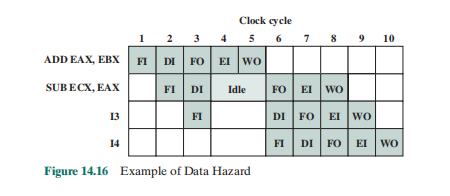
SUB ECX,EAX /\*ECX= ECX-EAX

第一条指令把32位寄存器EAX和EBX中的内容相加，把结果保存回EAX寄存器。第二条指令从ECX中减去EAX的值，并把结果保存回ECX。图14-16显示了流水线的行为。ADD指令在第5阶段结束之前才会更新寄存器EAX，这发生在时钟周期5，但是SUB指令在它执行的第2阶段就需要EAX的最新值，这发生在时钟周期4。要保证正确的操作，流水线必须停顿2个时钟周期。这样，在缺乏专门硬件和特殊的规避算法条件下，这一数据冒险导致流水线的运行效率降低。

有3种类型的数据冒险：

* **写后读（Read After Write， RAW）或真相关（true dependency）**：一条指令改写一个寄存器或内存位置，而后续的指令从所改写的寄存器或内存位置读取数据。如果在写操作完成之前，读操作就开始进行，那么就会发生冒险。
* **读后写（Write After Read， WAR）或反相关（antidependency）**：一条指令读一个寄存器或内存位置，而后续的指令将改写该寄存器或内存位置的内容。如果在读操作完成之前，写操作就开始进行，那么就会发生冒险。
* **写后写（Write After Write， WAW）或输出相关（output dependency）**：两条指令要改写同一个寄存器或内存位置。如果这两条指令的写操作发生次序与期望的次序相反，那么就会发生冒险。

图14-16显示了一个RAW型的冒险。另外两种类型的冒险在第16章有细致深入的讨论。



**3.控制冒险**

控制冒险（control hazard），又称为分支冒险（branch hazard），发生在流水线对分支转移做出了错误的预测，因此读取了在后期必须取消的指令之时。我们接下来讨论处理控制冒险的各种办法。

**处理分支指令**

设计指令流水线的一个主要问题是，保证有一个稳定的指令流进入流水线的最初几个阶段。正如我们已看到的，主要的障碍是条件分支指令。这种指令的特点是，直到指令实际被执行之前，不可能确定转移是否发生。

已有几种方法用于处理条件分支指令：

* 多个指令流（multiple streams）
* 预取分支目标（prefetch branch target）
* 循环缓冲器（loop buffer）
* 分支预测（branch prediction）
* 延迟分支（delayed branch）

**1.多个指令流**

一个简单的流水线之所以蒙受分支指令带来的惩罚，在于它必须在取下一条指令时做出二选一的选择，而且其选择可能是错的。一个强制的方法是复制流水线的开始部分，并允许流水线同时取这两条指令，使用两个指令流。这种方法有两个问题：

* 使用多个流水线，会有对寄存器和存储器访问的竞争延迟。
* 在原先的分支判断还没解决之前，可能又有另外的分支指令进入流水线（不管哪一路）。这样又需要添加指令流。

尽管有这些缺点，这个策略也能改善性能。使用两条或多条指令流水线的例子是IBM 370/168和IBM 3033机。

**2.预取分支目标**

识别出一个条件分支指令时，除了取此分支指令之后的指令外，分支目标处的指令也被取来。这个目标被保存直到分支指令被执行。若是分支发生，则目标已经被预取来了。

IBM 360/91使用这种方法。

**3.循环缓冲器**

循环缓冲器是由流水线指令取指阶段维护的一个小的但极高速的存储器，含有n条最近顺序取来的指令。若一个转移将要发生，硬件首先检查转移目标是否在此缓冲器中。若是，则下一条指令由此缓冲器取得。循环缓冲器有三个好处：

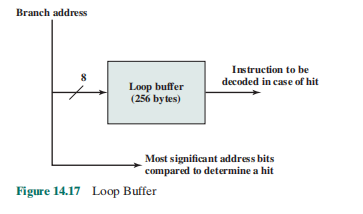
（1）采用指令预取，循环缓冲器将含有某些地址在当前指令地址之前的指令。于是，顺序取来的指令都可能被使用而不再需要通常的存储器访问时间。

（2）若一个转移的目标恰恰是在此转移指令之前的少数几个位置上，则目标已在缓冲器中。这对于相当普遍的IF-THEN和IF-THEN-ELSE序列特别有用。

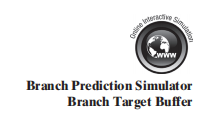
（3）这一策略非常适合处理循环或迭代，因此命名为循环缓冲器。若循环缓冲器充分大，足以容纳循环的全部指令，则这些指令只需要第一次循环时由存储器取来，后面的循环所需的指令都已经在缓冲区中，不需要再取指令。

从原理上讲，循环缓冲器类似于指令高速缓存。不同在于，循环缓冲器只保留顺序的指令，因而容量较小，成本也较低。

图14-17给出一个循环缓冲器的例子。若此缓冲器器容纳256字节，并且使用字节寻址，则转移地址的低8位可用于对缓冲器的索引。而其余的高有效位被检查，以确定转移目标是否已在此缓冲器所捕获的上下文中。



使用循环缓冲器的机器，有CDC的某些机器（Star-100、 6600、 7600）和CRAY-1。Motorola 68010使用了一种特殊形式的循环缓冲器，用于执行涉及DBcc （递减并依条件转移）指令的3指令循环（见习题14.14）。此时维持着一个3字缓冲器，处理器重复执行这些指令直到满足循环结束条件。



**4.分支预测**

已有几种不同技术用于预测转移是否将发生。其中应用最普遍的是：

* 预测绝不发生（ predict never taken）
* 预测总是发生（predict always taken）
* 依操作码预测（predict by opcode）
* 发生/不发生切换（taken/ not taken switch）
* 转移历史表（branch history table）

前三种方法是静态的，它们不取决于条件转移指令的过去执行历史。后两种方法是动态的，它们取决于执行的历史。

前两种方法最简单，它们或者总是假定转移不发生而继续顺序取指令，或者总是假定转移发生而从转移目标处取指令。其中预测绝不发生转移时所有分支预测方法中最广泛使用的方法。

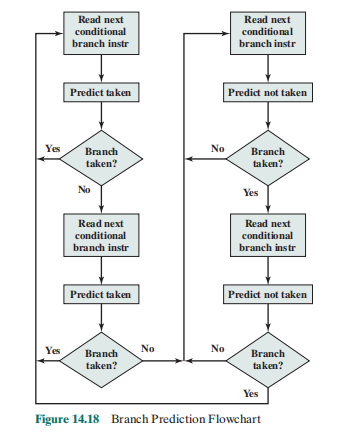
分析程序行为的研究已说明，条件分支的转移发生概率高于50% [LILJ88]。于是，若由每条路径预取的代价都是相同的，那么总是由转移目标地址的预取，应当比总是由顺序路径预取能给出更好的性能。然而，在一个分页的机器中。由转移目标的预取要比顺序预取下一条指令更可能引起缺页，故必须考虑到这种性能的损害。可使用一种规避机制来减少这种损失。

最后一种静态方法是依据转移指令的操作码进行判定。处理器假定，对某些条件分支指令的操作码将总是发生转移，对另外的总是不发生转移。[LILJ88]报告这种策略的成功概率大于75%。

动态分支预测策略，试图通过记录条件分支指令在程序中的历史来改善预测的准确度。例如，每个条件分支指令可有与之相关联的一位或几位，它们反映此指令的最近历史。这些位称为发生/不发生切换，它们指挥处理器在下次遇到此指令时产生具体的判定。一般，这些位不是保存在主存中，而是保存在一个暂时的高速存储装置中。一种可能是把这些位与对应的已在高速缓存（cache）中的条件分支指令相关联，当这种指令由cache替换出时其历史位也相应丢失。另一种可能是维护一个小型表，其中每个表项有最近执行的转移指令及其一位或几位相关位。处理器能像对待cache一样地访问这种关联表，或者通过使用分支指令地址的低序位来访问表。

以单个位所能记录的，只是这条指令最后一次执行是否发生转移。这种单个位方法的不足表现在像循环指令这样的几乎总是发生转移的条件分支指令使用时。对一个历史位，预测错误将出现两次，一次是进入循环时，一次是退出循环时。

若使用两位，则它们能用来记录相关指令的最后两次执行情况或是记录某种其他样式的状态。图14-18表示了一种典型的方法（另一种可能的样式见习题14.13）。假定算法由流程图的左上角开始。对一给定的条件分支指令，只要连续两次遇到的都是发生转移，则算法预测下一次转移要发生。如果一次预测失误，算法继续预测下一次转移将发生。仅当连续两次都不发生转移，算法才走到流程图的右边部分。接下来预测转移不发生，直到连续两次发生转移。于是，算法是连续两次失误才更改预测判定。

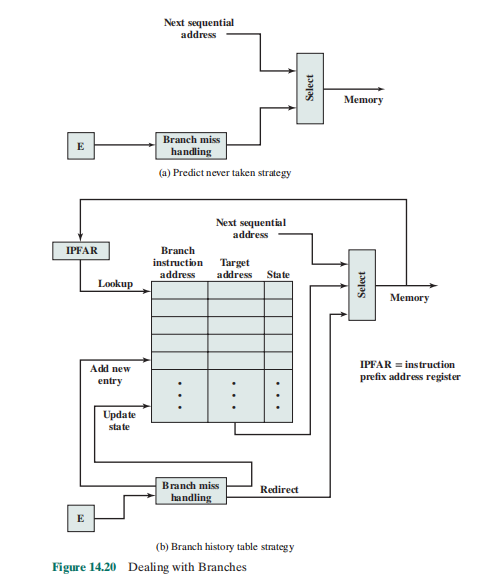
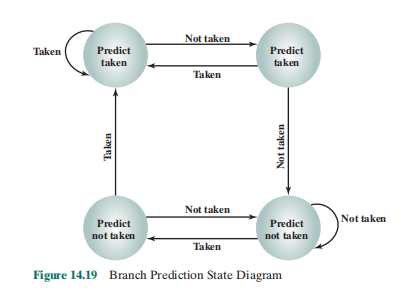


判定过程用有限状态机法能表示得更紧凑，如图14-19所示。有限状态机表示法为正式文献普遍采用。

使用刚才描述的历史位方法有一个缺点：若判定转移发生，转移目标指令并不能马上取得，直到作为条件分支指令操作数的目标地址被解析后才能取到。若判定一经做出，就能立刻开始取指令，可实现更高的效率。为此，必须保存更多的信息于一种称为分支目标缓冲器（branch target buffer）或分支历史表（branch history table）的结构中。

转移历史表是一个与流水线取指令阶段相关联的小型高速缓冲存储器。每个表项由三个元素组成：分支指令的地址，记录这条指令转移状况的历史位，有关它的目标指令的信息。在多数的建议和实现中，表项第三个字段保存的是目标指令的地址。另一种可能是直接保存目标指令。这里的权衡考虑是清楚的：保存目标地址可使表的规模较小，但与保存目标指令相比却要花费较多的取指令时间[RECH98]。

图14-20将这种策略与预测绝不发生策略做了对照。对后一种策略，取指阶段总是由顺序下一地址取指令。若一个转移发生，处理器中的某种逻辑检测到这个事件发生，于是指挥由目标地址处取下一指令（另外还要清空流水线）。转移历史表作为cache对待，每次预取触发一次转移历史表中的查找。若未发现匹配，则下一顺序地址用于取指。若发现匹配，依据指令状态进行预测；或是下一顺序地址，或是转移目标地址，将被送给选择逻辑。



当分支指令执行时，执行阶段将其结果通知转移历史表的控制逻辑。指令状态被修改以反映正确或不正确的预测。若预测不正确，则选择逻辑重定向到正确地址以便下次取指。当碰到的一个条件分支指令不在表中时，则它被添加到表中，而一个现有项被删除。这里会使用第4章讨论过的某种cache替换算法。

转移历史方法的一个改进版本是称为两级（two-level）或基于关联（correlation-based）的转移历史方法[YEH91]。这种方法基于如下假设：在循环结束处的分支，一个特定分支指令的过去历史，将是其未来行为的很好预测器。对于更为复杂的控制流结构，分支的方向经常与相关的分支指令方向有关。一个例子是if-then-else或case结构。可能采取的策略很多。一种典型的做法是，联合使用全局转移历史（即最近执行的分支指令的历史，而不仅仅是当前执行的分支指令历史）与当前分支指令的历史。通常的结构是定义为一个（m,n）关联预测器。该预测器使用最近m个分支指令的行为，从个n位分支预测器中选择一个，作为当前分支指令的预测器。换句话说，对于一个给定的分支指令，最近m条分支指令可能发生的每个组合都保留了一个n位历史记录。

5．延迟分支

改进流水性能的另一可能方法是自动重排程序中的指令，这样可以把一条分支指令移到实际所期望的位置之后。这种方法将在第15章讨论。

**Intel 80486的流水线**

一个有指导意义的指令流水线例子是Intel 80486处理器的指令流水线。80486处理器实现了一个5段流水线，如下所示：

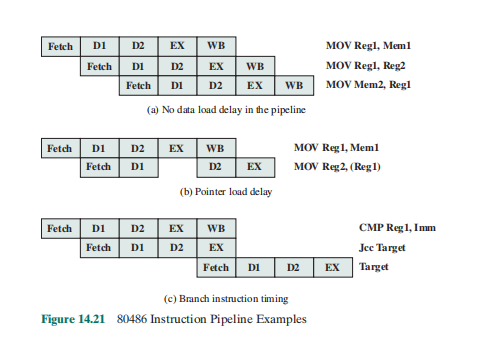
* **取指（fetch）**：指令由cache或外部存储器取来，并被放入两个16字节预取缓冲器中的某一个。取指阶段的目标是，只要旧的数据被指令译码器用掉，立即以新数据填充预取缓冲器。因为指令是可变长的（不计前缀， 1至11字节），故预取器的状态相对于流水线的其他阶段是由指令到指令变动的。平均而言，每个16字节的缓冲器大约装入5条指令[CRAW90]。取指阶段的操作独立于其他阶段以保持预取缓冲器的满载。
* **译码阶段1 （decode stage 1， D1 ）**：所有的操作码和寻址方式信息在D1阶段被译码，所要求的信息以及指令长度信息最多也只占据指令的前3个字节，于是， 3字节由预取缓冲器传送到D1阶段， D1译码器则能指挥D2阶段计算其余的信息（偏移量和立即数），这些是D1译码所不涉及的。
* **译码阶段2 （decode stage 2， D2）** ： D2阶段将每个操作码扩展成对ALU的控制信号。它还控制更复杂寻址方式的计算。
* **执行（execute， EX）**：这一阶段包括ALU运算， cache访问和寄存器修改。
* **写回（write back， WB）**：如果需要，这一阶段更改寄存器和在前面执行阶段修改过的状态标志。若当前指令要更改存储器，则计算结果值同时被送到cache和总线接口的写缓冲

中。

通过采用两个译码段，该流水线能支持接近每时钟周期一条指令的吞吐率。复杂指令和条件分支指令会降低这个乔吐率。

图14-21显示了此流水线操作的例子。图14-21a表示，当要求存储器访问时，并没有延时引人到流水线中，然而，正如图14-21b表示的那样，对于一个用于计算存储器地址的值，可能引入延时。也就是说，若一个值由存储器装入寄存器，而那个寄存器又要用作下一条指令的基址寄存器，则处理器要停顿一个时钟周期。在此例中，第一条指令的EX阶段处理器访问cache ，然后在WB阶段将所取得的值存入寄存器。然而，下一条指令在它的D2阶段就需要这个寄存器的值。当它的D2阶段与前一指令的WB阶段对齐时，一个旁路信号通路允许D2阶段去访问正被WB阶段所使用的同一数据，这样节省了一个流水段时间。

图14-21c说明了一条分支指令的时序。假定转移发生，比较指令在其WB阶段更新条件码，一个旁路信号通路允许分支指令的EX阶段能同时使用这些新条件码。在分支指令的EX阶段，处理器并行地运行一个猜测取指周期来读取转移目标指令。若处理器确认转移条件不成立，它将作废这个预取来的转移目标指令，并继续执行下一顺序指令（已被另一路预取缓冲器预先取来，并被译码）。



**14.5 x86系列处理器**

x86的组成结构随着时间不断发生显著变化。本节我们考察最新x86系列处理器组成的一些细节，并集中介绍各个处理器的共同结构要素。第16章会考察x86系列处理器超标量方面的特点。第18章考察多核组成结构。图4-18曾给出Pentium 4处理器组织结构的概要。

**寄存器组成**

x86系列处理器的寄存器组织包括如下类型的寄存器（见表14-2）。

* **通用寄存器（general）**： x86有8个32位通用寄存器（见图14-3c）。它们可由所有类型的x86指令使用，也可用来保存用于地址计算的操作数。此外，某些寄存器亦服务于专门目的。例如，字符串指令使用ECX、 ESI和EDI寄存器的内容作为操作数，无需指令中有对这些寄存器的显式引用。这样做的结果是，某些指令能更紧凑地编码。在64位模式中，有16个64位的通用寄存器。
* **段寄存器（segment）** ： x86有6个16位段寄存器，正如第8章所讨论的，它们容纳索引段表的段选择符。代码段（CS）寄存器引用含有正被执行指令的段。栈段（S5）寄存器引用含有用户可见栈的段。其余的段寄存器（DS， ES，FS， GS）允许用户一次可访问多达4个分开的数据段。
* **标志寄存器（flag）** ： 32位的EFLAGS寄存器容纳条件码和各种模式位。在64位模式中，该寄存器被扩展到64位，并被称为RFLAGS寄存器。在当前的x86体系结构定义中，RFLAGS寄存器的高32位未被使用。
* **指令指针寄存器（ instruction pointer）**：存有当前指令的地址。

表14-2 x86处理器的寄存器

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a）32位模式的整数单位 | | | | |
| 类型 | 数目 | 长度（位） | | 目的 |
| 通用 | 8 | 32 | | 通用用户寄存器 |
| 段 | 6 | 16 | | 含有段选择符 |
| 标志（EFLAGS） | 1 | 32 | | 状态和控制位 |
| 指令指针 | 1 | 32 | | 指令指针 |
| b）64位模式的整数单位 | | | | |
| 类型 | 数目 | 长度（位） | | 目的 |
| 通用 | 16 | 32 | | 通用用户寄存器 |
| 段 | 6 | 16 | | 含有段选择符 |
| 标志（EFLAGS） | 1 | 64 | | 状态和控制位 |
| 指令指针 | 1 | 64 | | 指令指针 |
| c）浮点单元 | | | | |
| 类型 | 数目 | 长度（位） | 目的 | |
| 数值 | 8 | 80 | 保持浮点数 | |
| 控制 | 1 | 16 | 控制位 | |
| 状态 | 1 | 16 | 状态位 | |
| 标记字 | 1 | 16 | 标志数值寄存器的内容 | |
| 指令指针 | 1 | 48 | 指向被异常中断的数据 | |
| 数据指针 | 1 | 48 | 指向被操作数中断的数据 | |

还有一些寄存器专门用于浮点单元：

* 数值寄存器（numeric）：每个寄存器保存一个扩展精度的80位浮点数。这样的寄存器有8个，它们的功能像一个栈，指令集中有相应的压人和弹出指令可用于对它们进行操作。
* 控制寄存器（control）： 16位的控制寄存器含有控制浮点单元操作的控制位，包括舍入类型控制，单、双或扩展精度控制，以及禁止或允许各种异常条件的位。
* 状态寄存器（status）： 16位状态寄存器包含反映浮点单元当前状态的位，包括一个指向数值寄存器栈顶的3位指针、报告最后运算结果的条件码，以及异常标志。
* 标记字寄存器（tag word）：这个16位寄存器为8个浮点数值寄存器每个保留2位标记它们用于指示相应寄存器内容的属性，四种可能值是有效、零、特殊数（NaN、无穷、非规格化）和空。这些标记允许程序在不对寄存器中的实际数据进行复杂解析的情况下，检查数值寄存器中的内容。例如，进行现场切换时，处理器不需要保存那些标记为“空”的浮点寄存器。

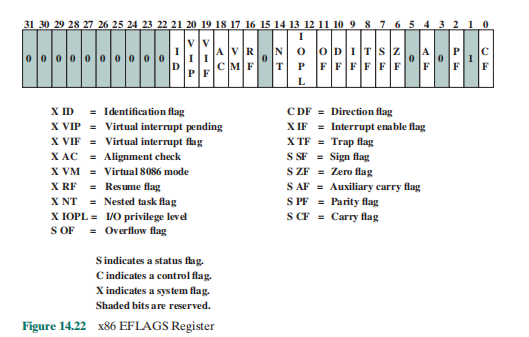
上述寄存器的大多数是很好理解的，下面简要地对几个寄存器做进一步说明。

**1. EFLAGS寄存器**

EFLAGS寄存器（参见图14-22）指出处理器的状态，并帮助控制处理器的操作。它包括表12-9所定义的6个条件码（进位、奇偶、辅助进位、零、符号、溢出），它们报告一次整数运算的结果。另外，此寄存器还有一些位可看作是控制位，它们是：

* **自陷标志（trap flag）**：当此TF位置位时，每条指令执行后都引起一个中断，这可用于调试。
* **中断允许标志（interrupt enable flag）** ：当此IF位置位时，处理器将响应外部中断。
* **方向标志（direction flag）** ：此DF位确定串行处理指令是递增还是递减16位寄存器SI和DI （16位操作）或32位寄存器ESI和EDI （32位操作）。
* **I/0特权级（I/O privilege lag）**：当此IOPL位置位时，保护模式期间所有对I/O设备的访问都将引起处理器产生一个异常。
* **重新开始标志（resume flag）** ：此RF位允许程序员禁止调试异常，这样在一次调试异常之后指令能重新开始，不会立即又引起另一次调试异常。
* **对齐检查（alignment check）**：若一个字或双字被寻址在一个非字或非双字边界上，此AC位将被激活。
* **标识标志（identification flag）** ：若此ID位能置位和复位，则表明这个处理器支持处理器ID指令。这条指令能提供处理器的有关厂商、系列、型号等信息。

此外，还有4位标志位涉及操作模式。嵌套任务（NT）标志指示，保护模式下的当前任务嵌套在另一任务中。虚拟模式（VM）位使程序员能允许或禁止虚拟8086模式，在这种模式下处理器作为一个8086机器来运行。虚拟中断标志（VIF）和虚拟中断未决标志（VIP）用于多任务环境。

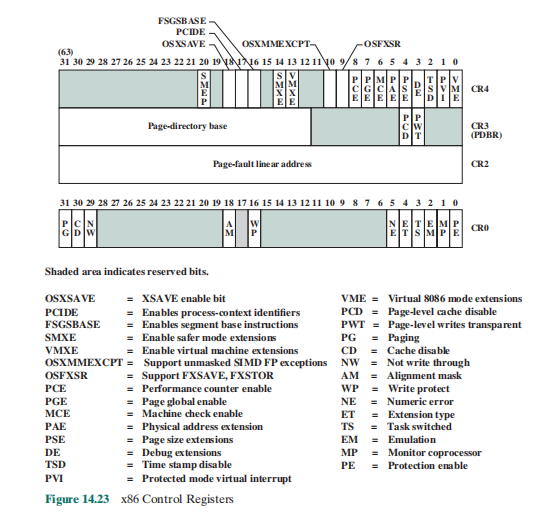


**2.控制寄存器**

x86使用了4个32位控制寄存器（寄存器CR1不使用）来控制处理器操作的各个方面（参见图14-23）。除了CR0寄存器之外，其余寄存器都可以是32位或64位宽度，具体位宽根据是否支持x86的64位体系结构而定。CR0寄存器包含系统控制标志，这些标志通常是控制处理器工作模式或指示其工作状态，而不是控制个别任务执行的。标志包括：

* **保护模式使能（ protection enable）**：此PE位允许/禁止保护工作模式。
* **监控协处理器（ monitor coprocessor）**：只有在早期的x86机器上运行程序时才需要关注此MP位。该位与数字协处理器的使用有关。
* **模拟（emulation）**：当处理器不具有一个浮点单元时，此EM位置位，试图执行一条浮点指令时将引起一个中断。
* **任务切换（task switched）**：此TS位指出处理器具有切换的任务。
* **扩展类型（extension type）**：不在Pentium及后续机器上使用，在早先的机器上此ET位用于指示对数字协处理器指令的支持。
* **数值错误（numeric error）**：此NE位允许在外部数据总线上报告浮点错误的标准机制。
* **写保护（write protect）**：当此WP位被清除时，一个监管进程（supervisor process）可向用户级只读页写入。在某些操作系统中，此特征可用于支持进程生成。
* **对齐屏蔽（alignment mask ）**：此AM位允许/禁止对齐检查。
* **非写直通（not write through）**：此NW位选择数据cache的操作模式，当它置位时，禁止数据cache的写直通操作。
* **高速缓存禁止（cache disable）**：此CD位允许/禁止使用内部cache填充机制。
* **分页（paging）**：此PG位允许/禁止分页。

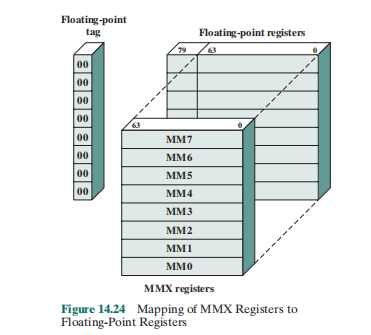
当允许分页时， CR2和CR3寄存器有效。CR2寄存器保存缺页中断之前最后访问页的32位线性地址。CR3最左20位保存页目录基地址的有效的高20位（低位为0不需保存）， CR3的两位用于驱动控制外部cache操作的引脚。页级高速缓存禁止（PCD）位允许或禁止外部cache。页级写直通（PWT）位控制外部cache的写直通。CR4中定义了其他的附加控制位。



**3. MMX寄存器**

回顾10.3节，我们知道x86的MMX功能利用了几种64位数据类型。MMX指令使用3位寄存器地址字段，故可以支持8个MMX寄存器的使用。实际上，处理器并未专门设置专用的MMX寄存器，而是使用了一种别名技术（aliasing technique），如图14-24所示，利用现有的浮点寄存器来保存MMX值。更准确地说，各浮点寄存器的低64位（尾数）用来构成8 个MMX寄存器。于是，现有的32位x86体系结构很容易扩展成支持MMX功能。MMX使用这些寄存器的关键特征如下所示。

* 回想一下，浮点寄存器在浮点运算中是作为栈来对待的。但对于MMX操作，同样这些寄存器却是直接可寻址的。
* 在任何浮点运算之后最初执行MMX指令时， FP的标记（tag）字段是有效的。这反映了由栈操作到直接寄存器寻址的改变。
* EMMS （清除MMX状态）指令设置FP标记字段的各位，令其指示所有寄存器都是空的。程序员在一个MMX代码块结束时插入这条指令是至关重要的，这样才能使后面的浮点运算正常运行。
* 当一个值写入MMX寄存器时， FP寄存器的[79： 64]位（符号和阶值位）被置为全1。这种设置使得以浮点数来看待这些寄存器时，这些FP寄存器的值是无穷大或NaN （非数），这保证了任何MMX数据值都不会被误认为是有效的浮点值。



**中断处理**

处理器中的中断处理是为支持操作系统提供的一种便利。它允许一个应用程序被挂起，以使各种中断事件能及时得到处理，然后再恢复应用程序的运行。

**1.中断和异常**

有两类事件能引起x86挂起当前指令流的执行并响应事件：中断（interrupt）和异常（exception）。在这两种情况下处理器都要保存当前进程的上下文，并将转至一个预先定义的子程序来执行特殊的服务。中断通常是由硬件信号产生的，并出现在程序执行期间内的任何时刻。异常是由软件产生的，由执行指令所引发。有两类中断源和两类异常源。

**（1）中断**

* **可屏蔽中断（maskable interrupt）**：由处理器的INTR引脚接收此信号。除非中断允许标志（IF）被置位，否则处理器不响应可屏蔽中断。
* **不可屏蔽中断（nonmaskable interrupt）**：由处理器的NMI引脚接收其信号。这类中断的响应不能被阻止。

**（2）异常**

* **处理器检测的异常（processor-detected exceptions）**：当试图执行一条指令而处理 器遇到一个错误时此异常发生。
* **程序异常（programmed exceptions）**：有一些指令（INTO、INT3、 INT和BOUND） 能产生异常。

**2.中断向量表**

x86的中断处理使用了中断向量表（interrupt vector table）。每一类中断都被指派了一个中断号，此号用于对中断向量表的索引。该表包含有256个32位中断向量，它们存储着中断服务程序的地址（段地址和偏移量）。

表14-3表示了中断向量号的指派情况，有阴影的项表示是中断，无阴影的项是异常。NMI硬件中断是类型2。INTR硬件中断号的范围在32-255。当一个INTR中断产生时，在总线上需要同时传送对应于此中断的中断向量号。其余的中断向量号用于异常。

表14-3 x86的异常和中断向量表

|  |  |
| --- | --- |
| 向量号 | 说明 |
| 0 | 除法错；除法上溢或被零除 |
| 1 | 调试异常；包括与调试有关的各种自陷和故障 |
| 2 | NMI引脚引发的中断；信号送至NMI引脚 |
| 3 | 断点；由INT3指令引起的，它是一条用于调试的1字节指令 |
| 4 | INTO检测到的上溢；处理器执行INTO的同时若OF标志置位将发生 |
| 5 | BOUND范围超出，BOUND指令比较一个寄存器的值与保存于存储器的边界值，若寄存器的值不在边界指定的范围内则产生一个异常 |
| 6 | 未定义的操作码 |
| 7 | 设备不可用；由于外部设备不存在，试图使用ESC或WAIT指令而失败 |
| 8 | 双重故障；在同一指令期间出现两个中断而且不能串行处理 |
| 9 | 保留 |
| 10 | 无效任务状态段；描述一个请求任务的段地址未被初始化或是无效 |
| 11 | 段不存在；要求的段不存在 |
| 12 | 栈故障；超过了栈段的界限或者栈段不存在 |
| 13 | 常规保护；不引起其他异常的保护违约（如向只读段的写） |
| 14 | 缺页 |
| 15 | 保留 |
| 16 | 浮点错；由浮点算术指令产生 |
| 17 | 对齐检查，以一个奇数字节地址存取一个字或以一个非4倍字节地址存取一个双字 |
| 18 | 机器检查，型号说明 |
| 19-31 | 保留 |
| 32-255 | 用户中断向量；当INTR信号激活时响应 |

注：不带阴影的是异常，有阴影的是中断。

若不止一个异常或中断是悬而未决的，则处理器以一个预先指定的顺序为它们服务。向量号在表中的位置不反映它们的优先级，异常和中断的优先级分为5类。以优先级降序排列的这5类优先级是：

* **类1**：先前指令上的中断（向量号1）；
* **类2**：外部中断（2， 32-255）；
* **类3**：取下一指令的故障（3， 14）；
* **类4**：下一指令的译码故障（6， 7）；
* **类5**：执行指令的故障（0， 4， 5， 8， 10-14， 16， 17）。

**3.处理中断**

正如使用CALL指令的转移执行流程一样，一个到中断处理子程序的控制转移也使用系统栈保存处理器的状态。当一个中断出现并被处理器响应时，如下事件序列发生：

（1）若转移涉及特权级改变，则当前栈段寄存器和当前扩展的栈指针（ESP）寄存器的内容被压人栈。

（2） EFLAGS寄存器的当前值被压人栈。

（3）中断（IF）和自陷（TF）两个标志被清除。这就禁止了INTR中断、自陷或单步中断。（4）当前代码段（CS）寄存器和当前指令指针（IP或EIP）寄存器的内容被压入栈。

（5）若中断伴随有错误代码，则错误代码也要压入栈。

（6）读取中断向量表对应项的内容，将其装入CS和IP （或EIP）寄存器。控制转移到中断服务子程序继续执行。

为从中断返回，中断服务子程序执行一条IRET指令。这使得所有保存在栈上的值被取回，并由中断点恢复执行。

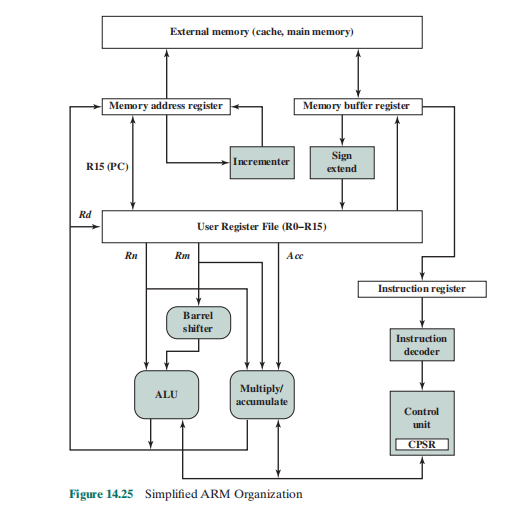
**14.6 ARM处理器**

本节我们考察ARM处理器的组成和体系结构中一些关键的要素。我们把其中较复杂的内容以及流水线组织安排到第16章进行讨论。对于本节和第16章的讨论，记住ARM体系结构的关建特征是有好处的。ARM处理器是一个RISC处理器，并有如下值得注意的特征：

* 中等规模、结构规整的寄存器组。寄存器数量比一些CISC机器多，但少于多数的RISC机器。
* 数据处理遵循装载/保存模式。其中各种运算只对寄存器中的操作数进行操作，而不直接访问内存。运算前，所需要的所有数据要先从内存装载到寄存器。运算结果可以继续被后继运算使用，或保存回存储器。
* 定长的、格式统一的32位标准指令集，以及一个16位的压缩指令集。
* 为使每条数据处理指令更为灵活，可以对一个源操作数进行移位或循环移位的预处理。为有效支持这一功能，设计了单独的算术逻辑单元和移位单元。
* 只提供了少数几种寻址模式，应用于所有装载/保存地址的确定。这些地址由指令中立即数或指定的寄存器操作数来计算得到。间接寻址，以及变址寻址因要用到内存中的值，故未被使用。
* 使用了自动递增和自动递减寻址模式，以便提高程序中循环的操作性能。
* 所有指令的执行都可带条件，这降低了条件分支指令的使用，从而减少了流水线清空，提高了流水线的效率。

**处理器组成**

ARM处理器的组织随着实现的不同相互之间差别较大，尤其当ARM处理器的实现是基于不同版本的ARM体系结构的时候。不过，对于本节的讨论，一个简单的一般化的ARM组织结构还是有用的，该组织结构如图14-25所示。图中，带箭头的线表示数据流。每个方框表示一个硬件功能单元或一个存储单元。



数据经由数据总线在处理器外的存储器和处理器之间传递。所传递的数据元素或者是装载/保存指令操作的数据项，或者是取指的指令。读取到的指令在控制单元的控制下，经过指令译码器，然后执行。控制单元包括流水线逻辑电路，并产生控制信号（图中未显示），送到处理器各个硬件单元。读取到的数据项放入由一组32位寄存器组成的寄存器组。以2的补码表示的字节或半字数据项会通过符号扩展到32位。

ARM的数据处理指令通常有两个源寄存器（Rn和Rm），一个目的或结果寄存器（Rd）。源寄存器值被送到算术逻辑单元，或单独的乘法单元进行计算。乘法单元中有额外的寄存器，以便累加部分积。ARM处理器还包含了一个单独的硬件单元，该单元可用于对源寄存器Rm的值在送到算术逻辑单元之前，做移位或循环移位操作。移位和循环移位操作可以在指令周期中完成这样能提高数据处理的能力和灵活性。

操作的结果会送回到目的寄存器中。装载/保存指令可能也会使用算术单元的结果来生成要装载或保存的存储器地址。

**处理器模式**

处理器只支持几种处理器模式是很常见的。例如，很多操作系统只使用两种模式：用户模式（user mode）和内核模式（kernel mode）。其中内核模式用来执行特权系统程序。ARM体系结构与上述方式不同，它提供了一个灵活的平台，以便操作系统实施不同的保护策略。

ARM处理器提供了7种运行模式。大多数应用程序在**用户模式**下运行。当处理器处于用户模式时，运行的程序不能访问受保护的系统资源，也不能改变模式，除非发生了异常。

其他6种运行模式称为特权模式，这些模式用于运行系统软件。定义这么多不同的特权模式有两个主要的好处： （1）操作系统可以对系统软件进行定制，以适应不同的情况； （2）特定的寄存器将用于特定的特权模式，使得上下文的切换更为便利。

异常模式可以访问所有的系统资源，并随意更改运行模式。特权模式中有5种是异常模式。当特定的异常发生时，就会进入到对应的异常模式。每种异常模式有一些专用的寄存器，它们取代了一些用户模式下的寄存器，这样做的目的是为了避免破坏在异常发生时的用户模式状态信息。异常模式如下所示：

* **监管模式（supervisor mode）**：这通常是操作系统运行的模式。当处理器碰到一条软件中断指令时，将进入这种模式。软件中断是ARM中一个调用操作系统服务的标准办法。
* **取消模式（abort mode）**：当出现内存错误时，将进入到这种模式。
* **未定义模式（undefined mode）**：当处理器试图执行一条既不被整数主处理核也不被协处理器支持的指令时，就进入到这种模式。
* **快速中断模式（fast interrupt mode）**：当处理器从指定的快速中断源接收到一个中断信号时，就进入到这种模式。快速中断服务程序是不能被中断的，但快速中断可以中断一个普通的中断服务程序。
* **中断模式（interrupt mode）**：当处理器从任何其他中断源（快速中断除外）接收到一个中断信号时，就进入到这种模式。只有快速中断可以中断一个中断服务程序。

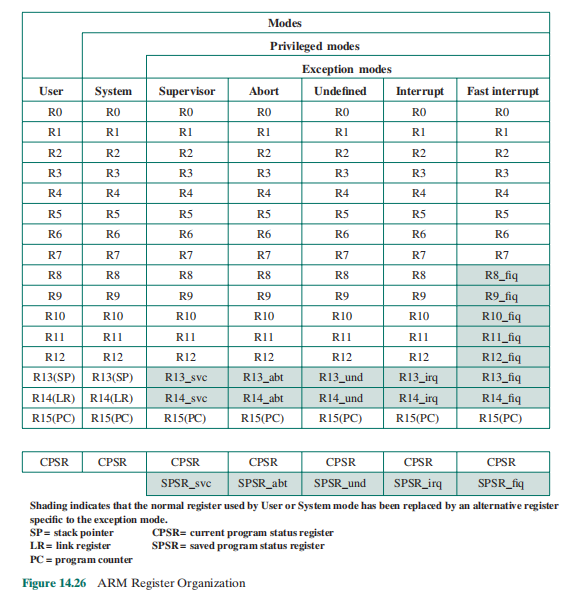
最后一种特权模式是**系统模式**，任何异常都不会进入这种模式，它与用户模式使用相同的寄存器。系统模式用于运行特定的特权操作系统任务，这些任务可以被上述5种异常模式中断。

**寄存器组成**

图14-26显示了ARM处理器对用户可见的寄存器。ARM处理器总计有37个32位的处理器，分类如下：

* + ARM处理器用户手册中介绍有31个通用寄存器。实际上，其中一些寄存器是有专门用途的，例如程序计数器。
  + 6个程序状态寄存器。

寄存器分成若干组，组与组之间有部分重叠。当前运行模式决定哪个组的寄存器是可见的。在任何时候，有16个寄存器， 1到2个程序状态寄存器是始终可见的，这样总计有17-18个软件可见的寄存器。下面给出图14-26的说明。



* 寄存器R0到R7，寄存器R15 （程序计数器），以及当前程序状态寄存器（CPSR），对所有模式可见，并被所有模式共享。
* 寄存器R8到R12由除了快速中断模式以外的模式共享。快速中断模式有它自己专用的寄存器R8\_fiq和R12\_fiq
* 所有的异常模式都有它们自己版本的寄存器R13和R14。
* 所有的异常模式都有它们自己专用的已存程序状态寄存器（SPSR）。

（1）通用寄存器：寄存器R13通常被用作栈指针，因此常常称为SP。因为每个异常模式下有自己单独的R13，因此每个异常模式都有自己专用的程序栈。R14被用作连接寄存器，用于保存子过程的返回地址以及异常模式退出时返回的结果。寄存器R15是程序计数器（PC）。

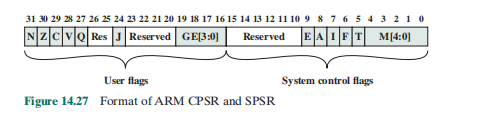
（2）程序状态寄存器：CPSR对于所有处理器模式都可访问。每个异常模式也有自己专用的SPSR寄存器。该寄存器用于保存异常发生时CPSR的值。

CPSR的高16位包含了用户模式下可见的那些标志。这些标志可以影响程序的操作（见图14-27）。下面列出了这些标志的说明：

* + **条件码标志**：第10章介绍过的N，Z，C和V标志。
  + **Q标志**：用于指示在一些面向SIMD的指令中是否发生了溢出和/或饱和（saturation）。
  + **J位**：表示现在在使用特殊的8位指令，即所谓的Java加速指令（Jazelle），这种指令的讨论不在本书范围内。
  + **GE [3：0]位**：SIMD指令使用bits [ 19：16]作为运算结果各个字节或半字的大于或等于（GE）标志。

CPSR的低16位包含了系统控制标志，它们只能在处理器处于特权模式时被修改。这些标志如下所示：

* **E位**：控制数据装载和保存的端序；对于指令读取，该位被忽略。
* **中断禁止位**： A位置位时，非精确地数据取消异常是被禁止的。I位置位时， IRQ中断被禁止。F位置位时，FIQ中断被禁止。
* **T位**：指示指令是否应该被当作普通的ARM指令，还是被当作压缩指令。
* **模式位**：指示当前处理器的运行模式。



**中断处理**

与其他处理器类似， ARM也提供了中断机制，允许处理器中断当前正在执行的程序，转而处理异常情况。异常由内部或外部的中断源产生，使得处理器应对某个事件。在处理异常前，处理器的状态通常要被保存，这样当异常处理完后，可以恢复执行被中断的程序。同一时间可以发生多于一个的异常。ARM处理器支持7种类型的异常，表14-4列出了这些异常类型，以及每种异常对应的处理器运行模式。当一个异常发生时，处理器的运行被强制转向到对应于该异常类型的某个内存固定地址处开始执行。这些固定的内存地址被称为异常向量。

表14-4 ARM中断向量

|  |  |  |  |
| --- | --- | --- | --- |
| 异常类型 | 运行模式 | 通常的入口地址 | 说明 |
| 重启 | 监管 | 0x00000000 | 当系统被初始化时发生 |
| 数据取消 | 取消 | 0x00000010 | 当访问一个无效内存地址时发生，例如一个地址没有对应的物理地址，或缺乏正确的访问许可 |
| 快速中断（FIQ） | 快速中断 | 0x0000001C | 当一个外部设备置位了处理器快速中断引脚时发生。一个中断处理程序一般不能被中断，除非该中断是一个快速中断。提供快速中断是为了支持数据传送或通道处理。快速中断模式提供了足够的私有寄存器，这样就不用考虑节省寄存器的使用，从而能使上下文切换的开销降至最低。快速中断服务程序是不能被中断的。 |
| 中断（FIQ） | 中断 | 0x00000018 | 当一个外部设备置位了处理器中断引脚时发生。中断服务程序不能被中断，除了快速中断以外 |
| 预取取消 | 取消 | 0x0000000C | 当试图读取一条指令，却导致内存错误时发生。该异常在指令进入流水线的执行阶段时被抛出 |
| 未定义指令 | 未定义 | 0x00000004 | 当一条不属于指令集的指令进入流水线的执行阶段时发生 |
| 软件中断 | 监管 | 0x00000008 | 通常用于允许用户模式的程序调用操作系统服务。在这种情况下，用户程序执行一条SWI指令，并带上相应参数，指出想要调用的系统服务 |

如果有多个中断等待处理，那么它们将按照优先级依次处理。表14-4就是按照异常的优先级次序从高到低排列的。

当异常发生时，处理器在执行完当前指令后，中止正在运行的程序。处理器的状态被保存到对应该异常的SPSR寄存器中。这样当异常处理程序执行完后，可以恢复原来程序的运行。处理器在异常发生时本来要执行的原程序的指令被保存在与该异常模式对应的连接寄存器（R14）中。异常处理完后，SPSR寄存器的内容将移到CPSR中，而R14的内容移到程序计数器中，从而返回原程序继续运行。

**12.7 关键词、思考题和习题**

**关键词**

branch prediction：分支预测

condition code：条件码

instruction eycle：指令周期

instruction pipeline：指令流水线

delayed branch：延迟分支

instruction prefetch：指令预取

Flag：标志

program status word：程序状态字

**思考题**

14.1 CPU寄存器通常起什么作用?

14.2 用户可见寄存器普遍支持的数据类型是什么?

14.3描述指令执行所必需的寄存器

14.4什么是程序状态字?

14.5 与不使用流水线相比，为什么一个两阶段流水线不可能将指令周期时间缩短到原来的一半?

14.6列出并简要说明指令流水线处理条件分支指令的几种方式。

14.7分支预测中如何使用历史位?

**习题**

14.1 （a）若在一个8位字的计算机上完成的最后操作是两个操作数00100100和00010001的加法，如下标志应该有何值?

·进位 ·符号 ·零 ·偶校验 ·上溢 · 半进位

（b）若两个操作数是-2（2的补码）和+3，又应该为何值?

14.2 若A含有11001100，B含有00110011，请对A-B操作重复习题14.1问题。

14.3 某微处理器的时钟频率是0.125ns：

（a）时钟周期是多长?

（b）由7个时钟周期组成的某特定类型的机器指令周期有多长时间?

14.4 某微处理器提供了能将字节串由内存一区域传送到另一区域的指令。取指令和最初译码用了10个时钟周期，此后每传送一字节用15个时钟周期。微处理器的时钟频率是10GHz. （a）请对32字节的串，确定指令周期长度。

（b）若此指令是不可中断的，那么最坏情况下中断响应的最大延迟是多少?

（c）若此指令在每字节传送开始前能被中断，重复（b）问。

14.5 Intel 8088由总线接口单元BIU和执行单元EU两部分组成，它们构成一个两段流水线。BIU负责取指令，放入一个4字节指令队列，并依EU请求参与地址计算、由内存取操作数和写回结果。若没有待处理的这类请求，并且总线空闲，则BIU填充指令队列的空位置。当EU完成一条指令的执行，它将结果传送给BIU （最终到存储器或I/O），然后再处理一下指令。

（a）假定BIU和EU完成各自的任务用相等的时间，流水线能提高8088性能多少倍?不考虑分支指令的影响。

（b）假定EU用时是BIU的两倍长，重复上问。

14.6 假定8088正在执行的程序中跳转指令出现的概率是0.1，为简化，认为所有指令都是2字节长。

（a）多大比率的指令读取总线周期被浪费了？

（b）若指令队列是8字节长，重复上问。

14.7 考虑图14-10的时序图。假定只是一个两阶段流水线（取指，执行）。重画此图，显示如果有4条指令的话，现在需要多少时间单位。

14.8 假定一流水有4段：取指（FI）、译码指令和地址计算（DA）、取操作数（FO）和执行（EX）。请为7条指令序列画出类似于图14-10的图，并假定此指令序列中的第3条指令是一条分支指令。另外，此序列不存在数据相关性。

14.9 某时钟速率为5GHz的流水式处理器执行一个有500万条指令的程序。流水线有4段并以每时钟周期2条的速率发射指令。不考虑分支指令和乱序（out-of-sequence）执行所带来的性能损失。

（a）同样执行这个程序，该处理器比非流水式处理器加速了多少?此处采用与14.4节相同的假设。

（b）此流水式处理器的吞吐率是多少（以MIPS为单位）?

14.10某时钟频率为10GHz的非流水式处理器，其平均CPI （每指令周期数）是5，此处理器的升级版本引入了4段流水。然而，由于如锁存延迟这样的流水线内部延迟，使新版处理器的时钟频率必须降到9GHz.

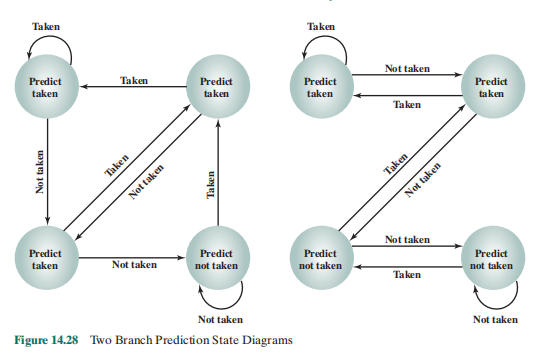
（a）同样执行这个程序，该处理器比非流水式处理器加速了多少?此处采用与14.4节相同的假设。

（b）此流水式处理器的吞吐率是多少（以MIPS为单位）?

14.11 考虑通过指令流水线来执行的一个长度为n的指令序列。假设遇到一条有条件的或无条件的分支指令的概念为p，并假设执行分支指令I，而转移到非后继连续地址的概率是q。为使问题简化，假设对于发生转移的分支指令I，当它在流水线最后一段执行时，将清空流水线并撤销线上其他正在进行的指令。请利用这些概率重写等式（14.1）和等式（14.2）。

14.12 对于处理流水线中分支指令的多个指令流方法，它的一个局限是，在最初分支指令是否转移还没确定之前又遇到另一个分支指令，请举出另外两个局限或缺点。

14.13 考虑图14-28所示的状态图。



（a）描述每个状态图的行为。

（b）将它们与14.4节分支预测的状态图做比较。讨论包括上述两种状态图在内的三种分支预测策略。

14.14 Motorola 680x0机器包括有“递减并根据条件转移” （decrement and branch according to condition）指令，它具有如下形式：

DBcc Dn， displacement

这里的cc是一个可测试条件，Dn是一个通用寄存器，displacement （偏移量）则指定相对于当前指令地址的目标地址。此指令能定义成：

if （cc =False）

then begin

Dn ：= （Dn）-1：

if Dn1 then PC：= （PC） + displacement end

else PC：=（PC） +2；

当指令执行时，首先测试条件以确定循环结束条件是否被满足。若是，则不执行任何操作，并继续执行顺序的下一条指令。若条件是假，则指定的数据寄存器被减1，并检查其值是否小于零。若是小于零，则循环结束，并继续执行顺序的下一条指令。否则，程序转移到指定的位置。现考虑如下的汇编语言程序段：

AGAIN CMPM.L （A0）+,（A1） +

DBNE D1, AGAIN

NOP

其中A0和A1是两个字串地址，代码对这两个串做比较，检查它们是否相等；每次访问了两个串中的对应元素，串指针都被递增。D1最开始含有待比较的长字（4字节）的数量。

（a）寄存器的初始值是： A0 = $00004000， A1 = $ 00005000，D1 = $000000FF（$表示十六进制数）。地址$4000和$6000间的存储器全部以$AAAA字装入。若运行上述程序，请指出DBNE循环执行的次数和当达到NOP指令时三个寄存器的内容。

（b）重复（a），但现在是假定存储器$4000和$4FEE之间是以 $0000字装入，而$5000和 $6000之间是以$AAAA装入。

14.15假定条件转移未发生，请重画图14-19c。

14.16摘自[MACD84]的表14-5对各类应用的转移行为进行了统计。除1类转移行为外，各类应用之间没有明显不同。请确定科学应用环境中，转向转移目标地址的转移占全部转移的比率。对于商业应用和系统应用环境，重复上一问题。

表14.5 示例应用程序中的转移行为

|  |  |  |  |
| --- | --- | --- | --- |
| 转移出现的类型：  1 类：转移指令 72.5%  2 类：循环控制 9.8%  3 类：过程调用，返回 17.7% | | | |
| 1 类转移：转向何处  无条件——100%转向目标  条件——转向目标  条件——不转向目标（顺序） | 科学  20%  43.2%  36.8% | 商业  40%  24.3%  35.7% | 系统  35%  32.5%  32.5% |
| 2 类转移（所有环境）  转向目标 91%  不转向目标 9% | | | |
| 3 类转移  100%转向目标 | | | |

14.17 流水化亦能施加到ALU内部以加速浮点运算。考虑浮点加减法的情况。简洁地说，流水线可以包含4段： （1）比较阶值；（2）选择阶值并对齐有效数；（3）加或减有效数； （4）规格化结果。假设流水有两个并行线程并能像这样着手进行：一个处理阶值，一个处理有效数。

图中标记R的方框指的是用于保持临时结果的一组寄存器。完善此框图，使其顶层视图表示出流水线的结构。

